

HoGent

BEDRIJF
EN
ORGANISATIE

Hoofdstuk 5 : LINQ

Hoofdstuk 5 : LINQ

- ▶ Inleiding
 - ▶ LINQ to Objects
 - Enkele eenvoudige LINQ methodes
 - Where en OrderBy operatoren
 - Select operatoren
 - Nog meer handige LINQ operatoren
 - ▶ Expression bodied members
 - ▶ Oefening
 - ▶ Appendix
- ▶ In dit hoofdstuk worden, naast LINQ, enkele nieuwe C# features geïntroduceerd die heel nauw aansluiten bij LINQ, kijk uit voor

C# feature ...

0. Repositories

- ▶ Theorie:
 - <https://github.com/WebIII/05thLinq>
- ▶ Oefeningen
 - <https://github.com/WebIII/05exLinq>
- ▶ Oplossingen
 - <https://github.com/WebIII/05solLinq>

LINQ

Inleiding

Inleiding

► LINQ: Language Integrated Query

- Querytaal
 - **SQL-like** queries in je C# code
- Compleet **geïntegreerd**
 - Intellisense, compile time checking van de queries, ...

- **A question:**
 - ◻ How much product did we sell ?
 - ◻ What is the average invoice amount per customer?
 - ◻ How many customers bought something this month?
- **A request:**
 - ◻ Get me the list of all customers that we invoiced this month along with the invoice amount
 - ◻ Get me the list of all past due customers
 - ◻ Get me a price list for all current products

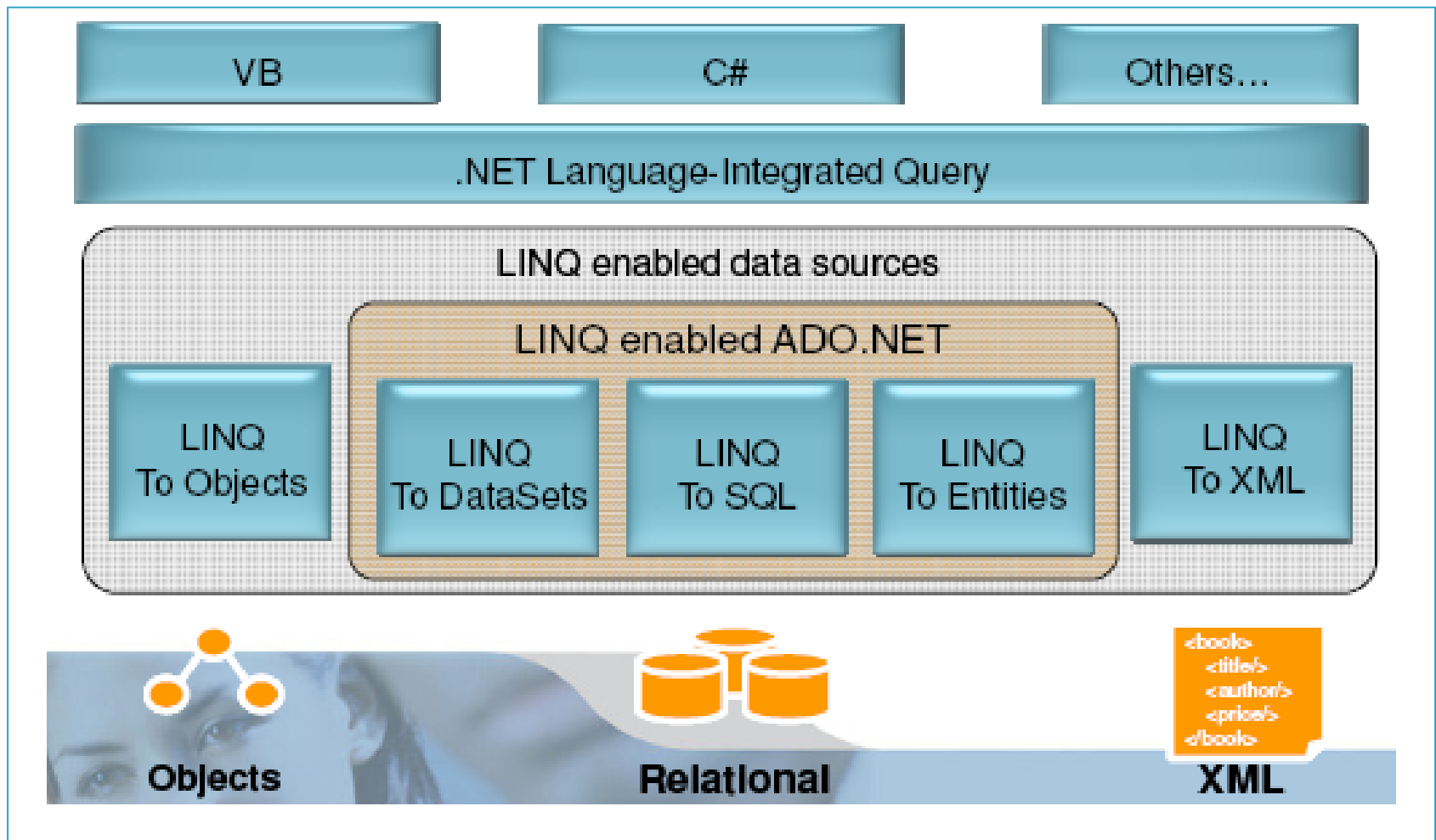


Inleiding

► Data source

- de data die bevraagd wordt
- heeft een LINQ-provider
- er zijn verschillende providers die toegang geven tot verschillende soorten data bronnen, bv.
 - **LINQ to objects** *(focus in dit hoofdstuk)*
 - bevragen van in memory data zoals strings, arrays, collections, ...
 - **LINQ to SQL**
 - bevragen van data in een relationele databank vanuit je C# code
 - **LINQ to entities** *(focus in later hoofdstuk)*
 - gelijkaardig maar gebruik makend van Entity Framework

Inleiding



Inleiding

- ▶ Eens je de LINQ syntax beheerst kan je op een uniforme manier werken met eender welke databron die een LINQ provider voorziet
 - LINQ to ...
 - Google, Twitter, eBay, Amazon, Flickr...
 - XML, JSON, ...
 - mySql, Oracle, ...
 - Excel, Word, ...
 - Javascript, ...
 - ...



Inleiding

- ▶ LINQ syntax
 - 2 verschillende soorten

gebruikt in deze cursus



QUERY syntax	METHOD syntax
<pre>var query = from c in customerList where c.CustomerId == customerId select c;</pre> <ul style="list-style-type: none">• Declaratief, ingebakken in C#• Wordt tijdens compilatie vertaald naar method syntax• Minder LINQ operatoren beschikbaar dan bij method syntax	<pre>var query = customerList.Where(c => c.CustomerId == customerId);</pre> <ul style="list-style-type: none">• Gebruik makend van methods• Onderdeel van .NET framework: System.Linq namespace in System.Core assembly

Inleiding

- ▶ Clone <https://github.com/WebIII/05thLinq.git>
- ▶ Wens je de code uit de slides te implementeren:
 - Open het View History venster in Team Explorer
 - Maak een nieuwe branch aan voor de Commit “Add Starter Application”

LINQ

Enkele eenvoudige LINQ methodes

Maak kennis met LINQ method syntax

C# feature ...

Extension methods en λ -expressies.

HoGent

► Extension methods – Wat?

- dit zijn methodes die toegevoegd worden aan een bestaande klasse om de **functionaliteit uit te breiden**
 - zonder overerving van de originele klasse
 - zonder de originele klasse zelf te wijzigen
 - dus zonder hercompilatie van de originele klasse
- **in gebruik verschillen ze niet van instance methods**

LINQ to Objects

► Extension methods – Hoe?

- een extension method declareer je als een **static method** in een **non generic static class**
- de eerste parameter in de parameterlijst laat je voorafgaan door het **keyword this**
 - het type van deze parameter is het type waarop je de extension method definieert

LINQ to Objects

► Extension methods - Voorbeeld

- Declaratie van een extension method die het type int uitbreidt:

Models/Extension.cs

```
namespace Extensions
{
    0 references
    public static class IntExtension
    {
        0 references
        public static bool IsEven(this int i)
        {
            return i % 2 == 0;
        }
    }
}
```

Zie Models folder
Extensions.cs en
step1.cs voor de
voorbeelden.

- Voorbeeld gebruik:

```
using Extensions;
```

Step1.cs

```
Console.WriteLine("Is 6 even? {0}", 6.IsEven());
Console.WriteLine("Is 7 even? {0}", 7.IsEven());
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("{0} is {1}", i, i.IsEven() ? "even" : "odd");
}
```

```
Is 6 even? True
Is 7 even? False
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

LINQ to Objects

► Extension methods - opmerkingen

- een extension method heeft **geen toegang tot private members** van de klasse waarop ze is gedefinieerd
- een extension method kan nooit een instance method 'overriden'
 - als een extension method dezelfde naam heeft als een instance method zal de compiler steeds de instance method kiezen
- extension methods zet je best in een aparte namespace
 - je moet die namespace expliciet in een using statement zetten om de extension methods in scope te brengen

LINQ to Objects

► Extension methods

◦ Voorbeeld 2:

Model/Extension.cs, class StringExtension

```
public static string RepeatText(this string s, int aantal)
{
    string resultaat = string.Empty;
    for (int i = 0; i < aantal; i++)
        resultaat += s;
    return resultaat;
}
```

Step1.cs

```
Console.WriteLine("6 times Hello!, that's {0}", "Hello!".RepeatText(6));
string myText = "Please repeat me...";
Console.WriteLine(myText.RepeatText(2));
```

```
6 times Hello!, that's Hello!Hello!Hello!Hello!Hello!Hello!
Please repeat me...Please repeat me...
```


LINQ to Objects

► Extension methods

◦ Voorbeeld 3:

Models/Extension.cs, class IntExtension

```
public static int CalculateSum(this IEnumerable<int> numbers)
{
    int aux = 0;
    foreach (int i in numbers)
    {
        aux += i;
    }
    return aux;
}
```

Step1.cs

```
IList<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Console.WriteLine(numbers.CalculateSum());
```

The sum is 15

deze extension method op `IEnumerable<int>` gaan we zelf niet schrijven, deze en veel meer **extension methods op `IEnumerable<T>`** vormen LINQ to objects...

LINQ to Objects

► Extension methods

◦ Oefening

- Vervolledig de extension method `IsDivisibleBy` in de klasse `IntExtension` in `Extension.cs`. Deze methode geeft aan of een geheel getal deelbaar is door een ander geheel getal (wordt als parameter opgegeven)
- Vervolledig de for loop in `Step1.cs` zodat enkel getallen tussen 1 en 20 die deelbaar zijn door 3 worden getoond, maak gebruik van bovenstaande extension method

LINQ

Bevragen van een in-memory data-source

Maak kennis met extension methods gedefinieerd op `IEnumerable<T>`

C# feature ...

Extension methods en λ -expressies.

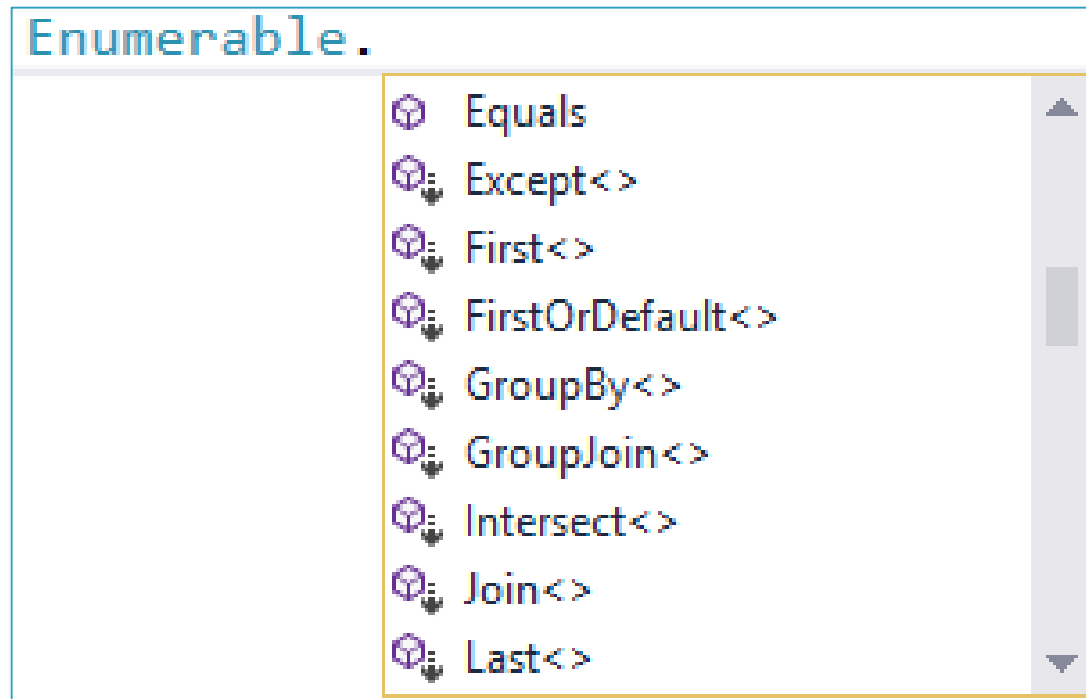
HoGent

LINQ to Objects

- ▶ **Bevragen van een in-memory data source**
 - LINQ methodes zijn **extension methods** gedefinieerd op **IEnumerable<T>**
 - Je kan ze gebruiken op elk type dat **IEnumerable<T>** implementeert, bv.
 - **Array**
 - **Generic collections**
 - **List<T>**, **Queue<T>**, **Stack<T>**, **HashSet<T>**, **LinkedList<T>**, **Dictionary<Tkey, Tvalue>**, **SortedList<Tkey, Tvalue>**, ...
 - De LINQ extension methods behoren tot de **namespace System.Linq**

LINQ to Objects

- ▶ **Bevragen van een in-memory data source**
 - **Enumerable** is een static class die alle LINQ extension methods bevat (this is van het type **IEnumerable<T>**)



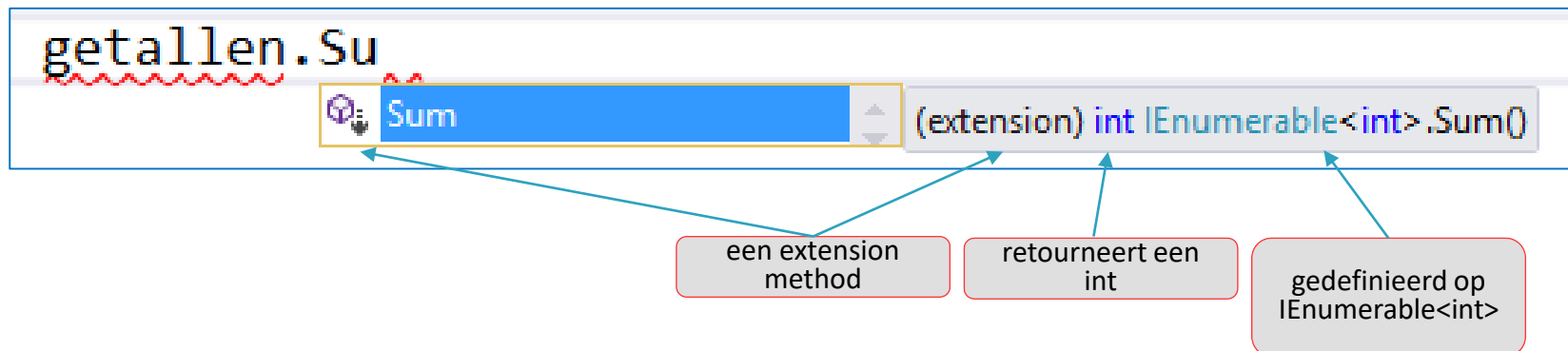
LINQ to Objects

► Sum()

- De **LINQ extension method Sum()** kan gebruikt worden op een collectie van getallen, ze retourneert de som van alle getallen in de collectie...
- **voorbeeld**

Step2.cs

```
int[] getallen = new int[] { 2, 8, 10 };  
int som = getallen.Sum();  
// som heeft de waarde 20
```



LINQ to Objects

► Sum()

- **Voorbeeld:** Sum() op enkele andere collecties...

Step2.cs

```
List<int> getallenLijst = new List<int> { 2, 8, 10 };
int somLijst = getallenLijst.Sum();
Console.WriteLine("De som van de getallen in de lijst is {0} ...", somLijst);

HashSet<double> getallenSet = new HashSet<double> { 2.5, 8.4, 10.6 };
double somSet = getallenSet.Sum();
Console.WriteLine("De som van de getallen in de hashset is {0} ...", somSet);

Stack<float> getallenStack = new Stack<float>();
getallenStack.Push(1.5F);
getallenStack.Push(2.6F);
float somStack = getallenStack.Sum();
Console.WriteLine("De som van de getallen op de stack is {0} ...", somStack);
```

```
De som van de getallen in de lijst is 20 ...
De som van de getallen in de hashset is 21,5 ...
De som van de getallen op de stack is 4,1 ...
```

LINQ to Objects

► Nog enkele eenvoudige LINQ methodes:


- **Average()**
- **Count()**
- **Min()**
- **Max()**

◦ Voorbeeld

Step2.cs

```
int[] getallen = new int[] { 2, 8, 10 };  
  
Console.WriteLine("Het gemiddelde is {0:0.00}", getallen.Average());  
Console.WriteLine("De collectie bevat {0} getallen", getallen.Count());
```

```
Het gemiddelde is 6,67  
De collectie bevat 3 getallen
```

 (extension) `int IEnumerable<int>.Count<int>()` (+ 1 overload)
Returns the number of elements in a sequence.

- **Oefening** : vervolledig opgaven in Step2.cs

LINQ

λ -expressies

Anonieme, inline functies ...

▶ λ -expressies

- **anonieme, inline** functies
 - maken gebruik van \Rightarrow , dit is de λ -operator
 - retourneren een waarde
- in LINQ maken we intensief gebruik van λ -expressies

LINQ to Objects

Models/Location.cs

```
public class Location
{
    public string Country { get; set; }
    public string City { get; set; }
    public int Distance { get; set; }

    public override string ToString()
    {
        return City + " in " + Country;
    }
}
```

► λ-expressies

- van gewone functies naar lambda's...

```
public int VoorbeeldFunctieMetNaam(Location loc)
{
    return loc.Distance;
}
```

- een C# functie heeft een type, het **type** is een **Func-delegate**
- je kan een functie dus toekennen aan een variabele van het type Func-delegate

```
private Func<Location, int> mijnFunctie = VoorbeeldFunctieMetNaam;
```

- via een lambda expressie kan dit alles echter kort en krachtig...

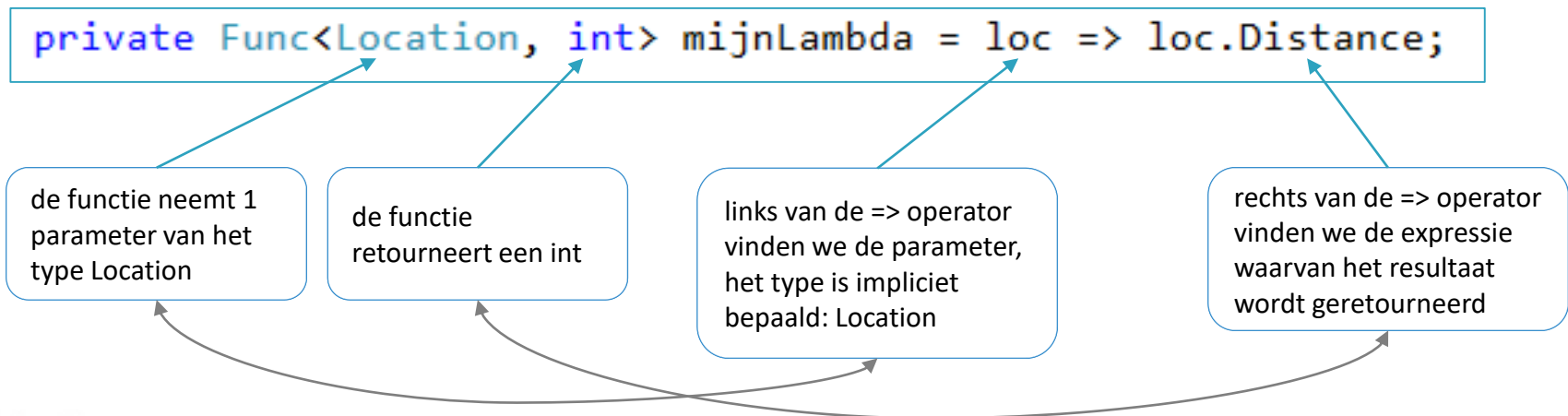
```
private Func<Location, int> mijnLambda = loc => loc.Distance;
```

de functie wordt **in-line** (on-the-fly) geschreven, zonder een voorafgaande declaratie;
de functie heeft geen naam, ze is **anoniem**;
de expressie is equivalent met onze oorspronkelijke functie

LINQ to Objects

► λ-expressies

- een lambda expressie is een anonieme functie van het type **Func<p1, p2, ..., pn, r>** waarbij
 - **p1, p2, ..., pn** de **types van de parameters** van de anonieme functie zijn, en
 - **r** het **returntype** van functie is
 - dit steeds het laatste type in de rij <p1, p2, ..., pn, r>



LINQ to Objects

▶ λ -expressie

- structuur wanneer er slechts 1 parameter is:

```
parameter => expression
```

- structuur wanneer er meerdere parameters zijn

```
(parameter1, parameter2, ... , parametern) => expression
```

LINQ to Objects

► λ-expressies

- aan methodes kunnen **λ-expressies als parameter** doorgegeven worden
- deze techniek wordt heel veel gebruikt bij LINQ
- **voorbeeld:** we willen de som van alle 'Distance'-s van een collectie van Locations

Models/TravelOrganizer.cs

```
public class TravelOrganizer
{
    // list of places visited and their distance to Seattle
    4 references
    public static IEnumerable<Location> PlacesVisited
    {
        get
        {
            return new List<Location>{
                new Location { City="London", Distance=4789, Country="UK" },
                new Location { City="Amsterdam", Distance=4869, Country="Netherland" },
                new Location { City="San Francisco", Distance=684, Country="USA" },
                new Location { City="Las Vegas", Distance=872, Country="USA" },
                new Location { City="Boston", Distance=2488, Country="USA" },
                new Location { City="Raleigh", Distance=2363, Country="USA" },
                new Location { City="Chicago", Distance=1733, Country="USA" },
                new Location { City="Charleston", Distance=2421, Country="USA" },
                new Location { City="Helsinki", Distance=4771, Country="Finland" },
                new Location { City="Nice", Distance=5428, Country="France" },
                new Location { City="Dublin", Distance=4527, Country="Ireland" }
            };
        }
    }
}
```

LINQ to Objects

► λ-expressies

◦ voorbeeld vervolg: werking

Step3.cs

```
IEnumerable<Location> placesVisited = TravelOrganizer.PlacesVisited;  
int sumDistances = placesVisited.Sum(l => l.Distance);
```

- de **IEnumerable<Location>** wordt element per element overlopen: **foreach**
 - elk element is van het type Location
- voor elk **Location object** wordt de **lambda expressie** aangeroepen
 - de loop variabele is het argument van de lambda expressie
- de lambda expressie retourneert voor **elk location-object een int**
 - **l => l.Distance**
- de **som van deze int-s** is het resultaat van de LINQ expressie en wordt toegekend aan sumDistances...


LINQ to Objects

► λ-expressies

◦ voorbeeld vervolg, een blik op Intellisense

Step3.cs

```
IEnumerable<Location> placesVisited = TravelOrganizer.PlacesVisited;  
int sumDistances = placesVisited.Sum(1 => 1.Distance);
```

 (extension) `int IEnumerable<Location>.Sum<Location>(Func<Location, int> selector)` (+ 9 overloads)

Computes the sum of the sequence of `int` values that are obtained by invoking a transform function on each element of the input sequence.

◦ **Sum<Location>**

- is een generische versie van Sum, de type-parameter is Location
- is gedefinieerd op **IEnumerable<Location>**
- Sum<Location> heeft 1 parameter van het type **Func<Location, int>**
 - we kunnen dus elke functie die 1 Location parameter heeft, en een int retourneert doorgeven
 - **lambda expressies** laten toe dat we op een heel eenvoudige wijze een argument voor deze parameter kunnen voorzien

LINQ to Objects

► λ-expressies

◦ voorbeeld – vervolg: een blik op msdn uitleg

`Sum<TSource>(IEnumerable<TSource>, Func<TSource, Int32>)`

Computes the sum of the sequence of `Int32` values that are obtained by invoking a transform function on each element of the input sequence.



◦ **Sum<TSource>**


- een generic method Sum, met een type parameter TSource
- het is een extension method gedefinieerd op **IEnumerable<TSource>**
- deze extension method heeft een parameter van het type **Func<TSource, Int32>**
 - we kunnen aan deze methode dus een lambda meegeven die een TSource parameter heeft, en een int retourneert
- via de lambda wordt **elk element van TSource getransformeerd naar een Int32**
- de **som van deze int-s** is het resultaat van Sum<TSource>

LINQ to Objects

- ▶ **λ-expressies**
 - **Voorbeeld2: een overload van Count()**

Step3.cs

```
int[] getallen = new int[] { 2, 8, 10 };  
int aantal = getallen.Count(g => g > 5);
```

 (extension) `int IEnumerable<int>.Count<int>(Func<int, bool> predicate)` (+ 1 overload)
Returns a number that represents how many elements in the specified sequence satisfy a condition.


```
Console.WriteLine("Er zijn {0} getallen groter dan 5.", aantal);
```

Er zijn 2 getallen groter dan 5.

LINQ to Objects

► λ-expressies

◦ Voorbeeld2: vervolg: blik op Intellisense

 (extension) `int IEnumerable<int>.Count<int>(Func<int, bool> predicate) (+ 1 overload)`
Returns a number that represents how many elements in the specified sequence satisfy a condition.

Count<int> is
een extension
method

Count<int>
retourneert
een int

Count<int> is
gedefinieerd op
IEnumerable<int>

Dit is een
generische
methode Count,
met type
parameter int

Count<int> heeft 1
parameter van het type
Func<int, bool>

◦ Oefening :

- bereken het aantal results hoger of gelijk aan 10 (zie Step3.cs)

LINQ

WHERE & ORDERBY

Ontdek de kracht van LINQ en leer wat deferred execution van een query is...

LINQ to Objects

► WHERE, filteren van collecties

`Where<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)`

Filters a sequence of values based on a predicate.

- de return waarde is een `IEnumerable` met enkel die elementen uit de collectie die **voldoen aan het predikaat**
- **voorbeeld:** filteren van een collectie Location objecten

Step3.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
// steden waarvan de naam langer is dan 5 posities...  
IEnumerable<string> citiesWithLongNames = cities.Where(c => c.Length > 5);  
foreach (string city in citiesWithLongNames)  
    Console.WriteLine(city);
```

```
London  
Amsterdam  
San Francisco  
Las Vegas  
Boston  
Raleigh  
Chicago  
Charlestown  
Helsinki  
Dublin
```

LINQ to Objects

► WHERE

◦ voorbeeld – werking toeglicht

```
// steden waarvan de naam langer is dan 5 posities...  
IEnumerable<string> citiesWithLongNames = cities.Where(c => c.Length > 5);
```

- de **IEnumerable<string>** wordt element per element overlopen:
foreach
 - de loop variabele is van het type string
- voor elk element wordt de **lambda expressie** aangeroepen
 - de loop variabele is het argument van de lambda expressie
- alle elementen waarvoor het resultaat van de lambda expressie true oplevert worden samen in een **IEnumerable<string>** geretourneerd

LINQ to Objects

► Filteren en sorteren

◦ WHERE

- het predikaat kan bestaan uit **gelijk welke boolese uitdrukking** (gebruik `||`, `&&`, ...)
- werpt een **ArgumentNullException** wanneer de collectie null is

◦ voorbeeld 2:

Step3.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
// steden waarvan de naam langer is dan 5 posities en een a bevat...  
IEnumerable<string> citiesFiltered = cities.Where(c => c.Length > 5 && c.Contains('a'));  
foreach (string city in citiesFiltered)  
    Console.WriteLine(city);
```

```
Amsterdam  
San Francisco  
Las Vegas  
Raleigh  
Chicago  
Charlestown
```

LINQ to Objects

► Filteren en sorteren

◦ WHERE – Deferred Execution

het resultaat van de query wordt berekend wanneer er over de query variabele wordt geïtereerd (foreach), en **niet** op het moment dat een waarde wordt toegekend aan de query variabele!

Step3.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
//steden waarvan de naam langer is dan 5 posities...  
IEnumerable<string> citiesWithLongNames = cities.Where(c => c.Length > 5);  
  
cities[0] = "Oostende";  
  
foreach (string city in citiesWithLongNames)  
    Console.WriteLine(city);
```

declaratie van de
query variabele
citiesWithLongNames

hier is het query
resultaat nodig,
de query wordt
hier pas
uitgevoerd

```
Oostende  
Amsterdam  
San Francisco  
Las Vegas  
Boston  
Raleigh  
Chicago  
Charlestown  
Helsinki  
Dublin
```


LINQ to Objects

► Filteren en sorteren

◦ WHERE – Deferred Execution

- telkens de query wordt uitgevoerd kan het resultaat verschillen...
- het resultaat is gebaseerd op de toestand van de databron op het moment dat de query uitgevoerd wordt...

Step3.cs

declaratie
van de
query
variabele

uitvoering
van de
query

uitvoering
van de
query

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
//steden waarvan de naam langer is dan 5 posities...  
IEnumerable<string> citiesWithLongNames = cities.Where(c => c.Length > 5);  
  
cities[0] = "Oostende";  
Console.WriteLine("Eerste iteratie...");  
foreach (string city in citiesWithLongNames)  
    Console.WriteLine(city);  
  
cities[0] = "Brussel";  
Console.WriteLine("Tweede iteratie...");  
foreach (string city in citiesWithLongNames)  
    Console.WriteLine(city);
```

```
Eerste iteratie...  
Oostende  
Amsterdam  
San Francisco  
Las Vegas  
Boston  
Raleigh  
Chicago  
Charlestown  
Helsinki  
Dublin  
Tweede iteratie...  
Brussel  
Amsterdam  
San Francisco  
Las Vegas  
Boston  
Raleigh  
Chicago  
Charlestown  
Helsinki  
Dublin
```

LINQ to Objects

► Deferred vs Immediate Execution

- alle methods die niet expliciet een `IEnumerable<T>` (of `IOrderedEnumerable<T>`) retourneren volgen **immediate execution**, i.e. uitvoering van de query gebeurt op de plaats van declaratie
 - Statistische methodes die 1 waarde retourneren
 - **Sum, Count, Average ,...**
 - Conversie methodes die de `IEnumerable<T>` converteren (zie verder)
 - **ToList, ToArray, ...**
- alle methodes die een `IEnumerable<T>` (of `IOrderedEnumerable<T>`) retourneren volgen **deferred execution**, i.e. uitvoering gebeurt wanneer er over effectief over de collectie geïtereerd wordt
 - ‘standaard’ query methods
 - **Where, Select, OrderBy, ...**

LINQ to Objects

► Filteren en sorteren

- **ORDERBY** (deferred execution)
 - OrderBy
 - OrderByDescending
 - ThenBy
 - ThenByDescending
 - Reverse
- **Voorbeeld**

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
// alle steden gesorteerd op naam  
IEnumerable<string> orderedCities = cities.OrderBy(c => c);  
foreach (string city in orderedCities)  
    Console.WriteLine(city);
```

```
Amsterdam  
Boston  
Charlestown  
Chicago  
Dublin  
Helsinki  
Las Vegas  
London  
Nice  
Raleigh  
San Francisco
```

LINQ to Objects

► Chaining extension methods

- de aanroepen naar verschillende extension methods kan je aan elkaar rijgen

► Voorbeeld

Step3.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
// steden waarvan de naam langer is dan 5 posities, gesorteerd op naam...  
IEnumerable<string> myCities = cities.Where(c => c.Length > 5).OrderBy(c => c);  
foreach (string city in myCities)  
    Console.WriteLine(city);
```

```
Amsterdam  
Boston  
Charlestown  
Chicago  
Dublin  
Helsinki  
Las Vegas  
London  
Raleigh  
San Francisco
```

LINQ to Objects

- ▶ Chaining extension methods
- ▶ voorbeeld 2

Step3.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin",  
    "San Anselmo", "San Diego", "San Mateo", "San Dimas"};  
  
// gefilterde en gesorteerde steden  
IEnumerable<string> orderedCities = cities.  
    Where(c => c.StartsWith("S")).  
    OrderByDescending(c => c.Length).  
    ThenBy(c => c);  
foreach (string city in orderedCities)  
    Console.WriteLine(city);
```

```
San Francisco  
San Anselmo  
San Diego  
San Dimas  
San Mateo
```

LINQ

SELECT

Leer hoe je collecties kunt omvormen tot andere collecties...

C# feature ...

Impliciete typering, anonieme types,
object & collection inializers.

LINQ to Objects

► SELECT

```
Select<TSource, TResult>(IEnumerable<TSource>,  
Func<TSource, TResult>)
```

Projects each element of a sequence into a new form.

- laat je toe elk element van een collectie te **transformeren naar een nieuw type**, dit type
 - kan eventueel gelijk zijn aan het originele type
 - kan een bestaand type zijn
 - kan een anoniem type zijn

LINQ to Objects

► SELECT

◦ voorbeelden

Step4.cs

```
int[] getallen = new int[] { 2, 8, 10 };  
IEnumerable<int> nieuweGetallen = getallen.Select(g => g + 1);  
foreach (int i in nieuweGetallen)  
    Console.WriteLine(i);
```

3
9
11

Step4.cs

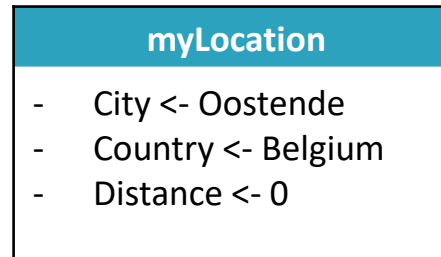
```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };  
  
// IEnumerable<string> wordt omgezet naar een IEnumerable<int>  
IEnumerable<int> citieLengths = cities.Select(c => c.Length);  
foreach (int cityLength in citieLengths)  
    Console.WriteLine(cityLength);
```

6
9
13
9
6
7
7
11
8
4
6

► Object Initializers

- laten je toe waarden toe te kennen aan **properties** van een object, **tijdens de creatie** van het object
 - de betrokken properties moeten publiek toegankelijk zijn
- **voorbeeld**

```
Location myLocation = new Location
{
    City = "Oostende",
    Country = "Belgium"
};
```



```
public class Location
{
    public string Country { get; set; }
    public string City { get; set; }
    public int Distance { get; set; }

    public override string ToString()
    {
        return City + " in " + Country;
    }
}
```

er wordt een instantie van type Location gemaakt a.d.h.v. de default constructor, tijdens creatie krijgen de properties City en Country expliciet een waarde toegekend

► Collection initializers

- laten je toe op een eenvoudige manier collecties te **instantiëren en te seeden**
 - werkt op een klasse die IEnumerable implementeert,
 - of een klasse die een Add-extension method voorziet
 - je kan de collectie seeden door gebruik te maken van simpele waarden, expressies of object initializers...
- **voorbeeld**

```
IList<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

IList<Location> myLocations = new List<Location> {
    new Location { City = "Oostende", Country = "Belgium"},
    new Location { City = "Amsterdam", Country = "Netherlands"},
    new Location { City = "Berlin", Country = "Germany"}
};
```

Collection initializer

Collection initializer, in
combinatie met object
initializer

LINQ to Objects

► SELECT

- **Voorbeeld:** omzetten van een collectie van Location objecten naar een collectie van CityDistance objecten...

```
public class Location
{
    public string Country { get; set; }
    public string City { get; set; }
    public int Distance { get; set; }

    public override string ToString()
    {
        return City + " in " + Country;
    }
}
```

```
public class CityDistance
{
    public string Country { get; set; }
    public string Name { get; set; }
    public int DistanceInKm { get; set; }

    public override string ToString()
    {
        return Name + " in " + Country;
    }
}
```

```
public class TravelOrganizer
{
    // list of places visited and their distance to Seattle
    public static IList<Location> PlacesVisited
    {
        get
        {
            return new List<Location>{
                new Location { City="London", Distance=4789, Country="UK" },
                new Location { City="Amsterdam", Distance=4869, Country="Netherland" },
                new Location { City="San Francisco", Distance=684, Country="USA" },
                new Location { City="Las Vegas", Distance=872, Country="USA" },
                new Location { City="Boston", Distance=2488, Country="USA" },
                new Location { City="Raleigh", Distance=2363, Country="USA" },
                new Location { City="Chicago", Distance=1733, Country="USA" },
            };
        }
    }
}
```

LINQ to Objects

► SELECT

◦ Voorbeeld vervolg

Step4.cs

```
IEnumerable<Location> placesVisited = TravelOrganizer.PlacesVisited;  
IEnumerable<CityDistance> cityDistances = placesVisited.Select(  
    l => new CityDistance  
    {  
        Name = l.City,  
        Country = l.Country,  
        DistanceInKm = (int)(l.Distance * 1.61)  
    });  
foreach (CityDistance c in cityDistances)  
    Console.WriteLine(c);
```

◦ werking

- de **IEnumerable<Location>** wordt overlopen: **foreach**
- voor elk Location object wordt de **lambda expressie** aangeroepen
 - het Location object is telkens het argument van de lambda expressie
 - de lambda expressie retourneert telkens een nieuw CityDistance object
- al deze objecten worden als **IEnumerable<CityDistance>** geretourneerd

► VAR

- voor variabelen gedeclareerd op method niveau (lokale variabelen) kan je als type **var** gebruiken
- hiermee introduceer je een **impliciet getypeerde variabele**
- dit is nog steeds een **sterk getypeerde variabele**, maar de compiler bepaalt zelf het type
 - je **moet** een impliciet getypeerde variabele **initialiseren bij declaratie**
 - **merk op:** Javascript var verschilt van deze C# var

```
var num = 50;  
var str = "simple string";  
var obj = new MyType();  
var numbers = new int[] { 1, 2, 3 };  
var dic = new Dictionary<int, MyType>();
```

compiler genereert
zelfde code

```
int num = 50;  
string str = "simple string";  
MyType obj = new MyType();  
int[] numbers = new int[] { 1, 2, 3 };  
Dictionary<int, MyType> dic = new Dictionary<int, MyType>();
```

LINQ to Objects

► SELECT (met gebruik van var)

◦ voorbeeld

Step4.cs

```
string[] cities = { "London", "Amsterdam", "San Francisco", "Las Vegas",  
    "Boston", "Raleigh", "Chicago", "Charlestown", "Helsinki", "Nice", "Dublin" };
```

```
// IEnumerable<string> wordt omgezet naar een type door de compiler bepaald...
```

```
var citieLengths = cities.Select(c => c + " " + c.Length);
```

(local variable) `IEnumerable<string>` citieLengths

```
foreach (var cityLength in citieLengths)
```

Console.Wri

(local variable) `string` cityLength

```
London 6  
Amsterdam 9  
San Francisco 13  
Las Vegas 9  
Boston 6  
Raleigh 7  
Chicago 7  
Charlestown 11  
Helsinki 8  
Nice 4  
Dublin 6
```

Via Intellisense kan je zien dat de compiler deze **var** vertaalt naar **string**, cityLength is sterk getypeerd...

► Anonieme types

- een **anoniem type** is een type die niet is benoemd, je introduceert het on-the-fly
 - het is een type **zonder klassedefinitie**
 - je maakt een nieuw object aan van het anonieme type door gebruik te maken van **new**, zonder type-specificatie
 - het type is bepaald door een **opsomming van properties**
 - deze properties zijn **read-only**

```
var homeTown = new
{
    Name = "Oostende",
    NrOfInhabitants = 60000
};
```

homeTown is een variabele die
impliciet getypeerd is (var)

homeTown wordt geïntanceerd **zonder een klassedefinitie**, het type van homeTown heeft geen naam, het **type is anoniem**, het type is volledig bepaald door de twee props name en nrOfInhabitants, homeTown is sterk getypeerd...

```
var homeTown = new
{
    Name
    NrOfInhabitants
};
```

(local variable) 'a' homeTown

Anonymous Types:
'a' is new { string Name, int NrOfInhabitants }

LINQ to Objects

► SELECT – Anonymous types

- in LINQ worden anonieme types dikwijls gebruikt om een collectie van objecten te transformeren naar een collectie van objecten die elk een subset van de properties van de originele objecten bevatten
- **Voorbeeld** omzetten van een collectie van Location objecten naar een collectie van anonieme objecten...

```
public class Location
{
    public string Country { get; set; }
    public string City { get; set; }
    public int Distance { get; set; }

    public override string ToString()
    {
        return City + " ";
    }
}
```

```
public class TravelOrganizer
```

```
{
    // list of places visited and their distance to Seattle
```

```
    public static IList<Location> PlacesVisited
```

```
{
```

```
    {
```

```
        get
```

```
        {
```

```
            return new List<Location>{
```

```
                new Location { City="London", Distance=4789, Country="UK" },
```

```
                new Location { City="Amsterdam", Distance=4869, Country="Netherlands" },
```

```
                new Location { City="San Francisco", Distance=684, Country="USA" },
```

```
                new Location { City="Las Vegas", Distance=872, Country="USA" },
```

```
                new Location { City="Boston", Distance=2488, Country="USA" },
```

```
                new Location { City="Raleigh", Distance=2363, Country="USA" }
            }
        }
    }
}
```


LINQ to Objects

► SELECT – Anonymous types

◦ Voorbeeld vervolg

Step4.cs

```
IEnumerable<Location> placesVisited = TravelOrganizer.PlacesVisited;  
var anonymousCities = placesVisited.Select(c => new  
{  
    Name = c.City,  
    DistanceInKm = c.Distance * 1.61  
});
```

de anoniem getypeerde objecten bevatten de naam van de stad en de afstand tot Seattle maar nu omgezet naar km

```
public class Location  
{  
    public string Country { get; set; }  
    public string City { get; set; }  
    public int Distance { get; set; }  
  
    public override string ToString()  
    {  
        return City + " in " + Country;  
    }  
}
```



```
new  
{  
    Name = l.City,  
    DistanceInKm = l.Distance * 1.61  
}
```

LINQ

Nog meer LINQ operatoren

Nog een selectie aan handige operatoren...

LINQ to Objects

► Nog meer handige LINQ methods

◦ **First()**

- retourneert het eerste element uit de collectie
- InvalidOperationException als collectie leeg is
- ArgumentNullException als collectie null is

◦ **FirstOrDefault()**

- retourneert het eerste element uit de collectie
- retourneert de default waarde als de collectie leeg is
 - dit is null voor nullable en reference types
- ArgumentNullException als collectie null is

◦ **Interessante overloads**

- First(predicate) / FirstOrDefault(predicate)

◦ **Last() / LastOrDefault()**

- volledig analoog

LINQ to Objects

- ▶ Nog meer handige LINQ methods
 - voorbeeld

```
int[] getallen = new int[] { 2, 8, 10 };  
// x krijgt de waarde 2  
int x = getallen.FirstOrDefault();  
// y krijgt de waarde 8  
int y = getallen.FirstOrDefault(g => g > 2);  
// z krijgt de waarde 0  
int z = getallen.FirstOrDefault(g => g < 2);  
// i krijgt de waarde 2  
int i = getallen.First();  
// j krijgt de waarde 8  
int j = getallen.First(g => g > 2);  
// er wordt een InvalidOperationException geworpen...  
int t = getallen.First(g => g < 2);
```

LINQ to Objects

► Nog meer handige LINQ methods

- **Skip()**

- slaat alle elementen uit de collectie over, tot een bepaalde positie, en retourneert dan de rest van de elementen

- **Take()**

- retourneert alle elementen van het begin van de collectie tot op een bepaalde positie in de collectie

- **SkipWhile/TakeWhile**

- analoog maar nu worden elementen overgeslaan/genomen tot we aan een element komen die aan een bepaald predikaat voldoet

- Deze methodes retourneren IEnumerable types en volgen dus **deferred execution**

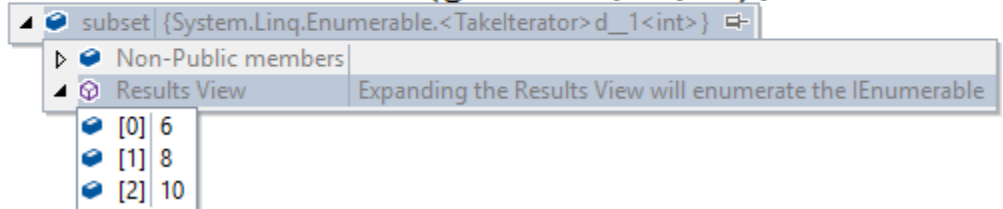
LINQ to Objects

► Nog meer handige LINQ methods

◦ voorbeeld Skip/Take

```
// Retourneert een subset van de getallen (1-based)
public static IEnumerable<int> GeefGetallen(IEnumerable<int> getallen, int van, int totEnMet)
{
    return getallen.Skip(van - 1).Take(totEnMet - van + 1);
}
```

```
int[] getallen = new int[] { 2, 4, 6, 8, 10, 12 };
// neem derde tot en met vijfde element uit de collectie
IEnumerable<int> subset = GeefGetallen(getallen, 3, 5);
```



LINQ to Objects


► Nog meer handige LINQ methods

◦ **SelectMany()**

- vormt elk element van een collectie om tot een IEnumerable,
- en plakt al deze IEnumerable's samen tot 1 IEnumerable ("flattening the result")

Step5.cs

```
IList<Location> myLocations = new List<Location> {  
    new Location { City = "Oostende", Country = "Belgium"},  
    new Location { City = "Amsterdam", Country = "Netherlands"},  
    new Location { City = "Berlin", Country = "Germany"}  
};  
  
var alles = myLocations.SelectMany(l => new List<string> { l.City, l.Country });
```

 (extension) IEnumerable<string> IEnumerable<Location>.SelectMany<Location, string>(Func<Location, IEnumerable<string>> selector) (+ 3 overloads)
Projects each element of a sequence to an IEnumerable<out T> and flattens the resulting sequences into one sequence.

```
foreach (var s in alles)  
    Console.WriteLine("- {0} -", s);
```

```
- Oostende -  
- Belgium -  
- Amsterdam -  
- Netherlands -  
- Berlin -  
- Germany -
```

LINQ to Objects

► Nog meer handige LINQ methods

◦ **GroupBy()**

- laat je toe elementen uit een collectie te groeperen
- het resultaat is een **IEnumerable** van **IGrouping**
 - elke **IGrouping** bevat een **Key** en een collectie van bijhorende objecten

Step5.cs

```
ICollection<Location> myLocations = new List<Location> {  
    new Location { City = "Oostende", Country = "Belgium"},  
    new Location { City = "Amsterdam", Country = "Netherlands"},  
    new Location { City = "Gent", Country = "Belgium"},  
    new Location { City = "Amersfoort", Country = "Netherlands"},  
    new Location { City = "Barcelona", Country = "Spain"}  
};  
  
var overzicht = myLocations.GroupBy(l => l.Country, l => l.City);  
  
foreach (var overzichtsGroep in overzicht)  
{  
    Console.WriteLine(overzichtsGroep.Key);  
    foreach (string city in overzichtsGroep)  
        Console.WriteLine("  - {0}", city);  
}
```

```
Belgium  
  - Oostende  
  - Gent  
Netherlands  
  - Amsterdam  
  - Amersfoort  
Spain  
  - Barcelona
```


LINQ to Objects

► Nog meer handige LINQ methods

◦ **ToList(), ToArray(), ToDictionary(), ...**

- vormt een IEnumerable om tot een lijst/array/dictionary/...
- deze conversie zorgt voor **immediate execution** van de query

```
ICollection<Location> myLocations = new List<Location> {  
    new Location { City = "Oostende", Country = "Belgium"},  
    new Location { City = "Amsterdam", Country = "Netherlands"},  
    new Location { City = "Gent", Country = "Belgium"},  
    new Location { City = "Amersfoort", Country = "Netherlands"},  
};  
  
var locsInBelgium = myLocations.Where(l => l.Country == "Belgium").ToList();  
  
myLocations.Add(new Location { City = "Brugge", Country = "Belgium" });  
  
Console.WriteLine("Cities in Belgium:");  
foreach (var l in locsInBelgium)  
{  
    Console.WriteLine(" - {0}", l.City);  
}
```

```
Cities in Belgium:  
- Oostende  
- Gent
```

de query wordt hier direct uitgevoerd want we gebruiken .ToList()

zonder .ToList() zou Brugge wel deel van de uitvoer zijn...

LINQ to Objects

► Nog meer handige LINQ methods

◦ All()

<code>All<TSource></code>	Determines whether all elements of a sequence satisfy a condition.
---------------------------------	--

◦ Any()

<code>Any<TSource>(IEnumerable<TSource>)</code>	Determines whether a sequence contains any elements.
<code>Any<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)</code>	Determines whether any element of a sequence satisfies a condition.

◦ Distinct()

<code>Distinct<TSource>(IEnumerable<TSource>)</code>	Returns distinct elements from a sequence by using the default equality comparer to compare values.
<code>Distinct<TSource>(IEnumerable<TSource>, IEqualityComparer<TSource>)</code>	Returns distinct elements from a sequence by using a specified <code>IEqualityComparer<T></code> to compare values.

◦ Contains(), ElementAt(), en zo veel meer...

! zie !

[https://msdn.microsoft.com/en-us/library/vstudio/system.linq.enumerable_methods\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/vstudio/system.linq.enumerable_methods(v=vs.100).aspx)

LINQ

Expression bodied members

Expression-Bodied Function Members

- ▶ Ook in properties en methodes kan je gebruik maken van =>

Step6.cs

```
public class Person
{
    private string lName;

    public Person(string firstName, string lastName)
    {
        FName = firstName;
        LName = lastName;
    }

    public string LName
    {
        get => lName;
        set => lName = value;
    }

    public String FName { get; set; }

    public String FullName => $"{FName} {LName}";

    public override string ToString() => $"{FName} {LName}".Trim();

    public void DisplayName() => Console.WriteLine(ToString());
}
```

Getter only property

LINQ

Oefening

Oefening

- Oefening :
Zie **Step7.cs**

```
-----LINQ Menu-----
1. Step 1: Extension methods.
2. Step 2: Enkele eenvoudige Linq operatoren.
3. Step 3: Where en lambda expressies
4. Step 4: Select en var/anonieme types
5. Step 5: Nog meer handige LINQ operatoren
6. Step 6: Expression bodied members
7. Oefeningen
99. Exit.
Enter your choice :
```

Step 7 : Exercises

```
Gemiddelde afstand van de steden is 3177 miles
De verste stad ligt op 5428 miles
De verste stad in de USA is Boston in USA at 2488 miles distance
```

```
----- Dichte steden buiten de USA -----
Ireland
```

```
----- Alle landen in de lijst: -----
Finland
France
Ireland
Netherland
UK
USA
```

Welke steden liggen in de USA?

```
----- Anonieme type voor landen: -----
{ City = London, InUSA = False }
{ City = Amsterdam, InUSA = False }
{ City = San Francisco, InUSA = True }
{ City = Las Vegas, InUSA = True }
{ City = Boston, InUSA = True }
{ City = Raleigh, InUSA = True }
{ City = Chicago, InUSA = True }
{ City = Charleston, InUSA = True }
{ City = Helsinki, InUSA = False }
{ City = Nice, InUSA = False }
{ City = Dublin, InUSA = False }
```


```
----- CityDistances voor steden in USA: -----
San Francisco in USA at 1094 km distance
Las Vegas in USA at 1395 km distance
Boston in USA at 3980 km distance
Raleigh in USA at 3780 km distance
Chicago in USA at 2772 km distance
Charleston in USA at 3873 km distance
```

Appendix

- de ForEach() method
- reflection in C#

Appendix : ForEach() method

- Op List<T> is **de methode ForEach()** gedefinieerd

	ForEach(Action<T>)	Performs the specified action on each element of the List<T>.
---	--------------------	---

- Voorbeeld:

Action<T> is een anonieme functie met T als parameter en die void retourneert.

```
IEnumerable<Location> placesVisited = TravelOrganizer.PlacesVisited;  
  
List<Location> allPlaces = placesVisited.ToList();  
allPlaces.ForEach(ap => Console.WriteLine(ap));
```

de lambda expression bevat een anonieme functie die void retourneert... en is van het type **Action<T>**

```
London in UK at 4789 miles distance  
Amsterdam in Netherland at 4869 miles distance  
San Francisco in USA at 684 miles distance  
Las Vegas in USA at 872 miles distance  
Boston in USA at 2488 miles distance  
Raleigh in USA at 2363 miles distance  
Chicago in USA at 1733 miles distance  
Charleston in USA at 2421 miles distance  
Helsinki in Finland at 4771 miles distance  
Nice in France at 5428 miles distance  
Dublin in Ireland at 4527 miles distance
```


Appendix : Reflection

► Reflection



In object oriented programming languages such as [Java](#), reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods.

Reflection can also be used to adapt a given program to different situations dynamically. For example, consider an application that uses two different classes `X` and `Y` interchangeably to perform similar operations. Without reflection-oriented programming, the application might be hard-coded to call method names of class `X` and class `Y`. However, using the reflection-oriented programming paradigm, the application could be designed and written to utilize reflection in order to invoke methods in classes `X` and `Y` without hard-coding method names. Reflection-oriented programming

- EF en andere ORM tools maken daar gebruik van

Appendix : Reflection

► Voorbeeld : Main methode in Program.cs

```
if (keuze != "99")
{
    Type type = Type.GetType("Linq.Step" + keuze);
    if (type != null)
    {
        Object o = Activator.CreateInstance(type);
        type.GetMethod("Execute").Invoke(o, null);
    }
}
```

Type discovery : reflection zoekt een klasse in de assembly met de naam Linq.Step1.

Creëert een instantie van die klasse

Voert de methode Execute van dit object uit (null : daar deze methode geen parameters vereist)

Referenties

- ▶ ScottGu's Blog - Using LINQ with ASP.NET (Part 1). (n.d.). Retrieved August 07, 2014, from <http://weblogs.asp.net/scottgu/Using-LINQ-with-ASP.NET-2800-Part-1-2900>
- ▶ LINQ (Language-Integrated Query). (n.d.). Retrieved August 07, 2014, from <http://msdn.microsoft.com/en-us/library/bb397926.aspx>
- ▶ Uitgebreide lijst met LINQ voorbeelden op <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>
- ▶ **Pluralsight:**
 - LINQ Fundamentals with C# 6.0 by Scott Allen
 - Practical LINQ by Deborah Kurata
- ▶ Microsoft Virtual Academy :
 - Demystifying Linq : <https://mva.microsoft.com/en-US/training-courses/demystifying-linq-12301?l=94qlp9SKB-8804668937#>